CHAPTER 1

CHAPTER 1 Introduction

1.1 Overview

The Climate Data Management System is an object–oriented data management system, specialized for organizing multidimensional, gridded data used in climate analysis and simulation.

CDMS is implemented as part of the Climate Data Analysis Tool (CDAT), which uses the Python language. The examples in this chapter assume some familiarity with the language and the Python Numeric module (http:// numpy.sf.net). A number of excellent tutorials on Python are available in books or on the Internet. For example, see http://python.org .

1.2 Variables

The basic unit of computation in CDMS is the *variable*. A variable is essentially a multidimensional data array, augmented with a *domain*, a set of *attributes*, and optionally a spatial and/or temporal *coordinate system* (see Coordinate Axes on page 11). As a data array, a variable can be sliced to obtain a portion of the data, and can be used in arithmetic computations. For example, if **u** and **v** are variables representing the eastward and northward components of wind speed, respectively, and both variables are functions of time, latitude, and longitude, then the velocity for time 0 (first index) can be calculated as

```
>>> from cdms import MV
>>> vel = MV.sqrt(u[0]**2 + v[0]**2)
```

This illustrates several points:

- Square brackets represent the slice operator. Indexing starts at 0, so **u[0]** selects from variable **u** for the first timepoint. The result of this slice operation is another variable. The slice operator can be multidimensional, and follows the syntax of Numeric Python arrays. In this example, **u[0:10,:,1]** would retrieve data for the first ten timepoints, at all latitudes, for the second longitude.
- Variables can be used in computation. '**' is the Python exponentiation operator.
- Arithmetic functions are defined in the **cdms.MV** module.
- Operations on variables carry along the corresponding metadata where applicable. In the above example, **vel** has the same latitude and longitude coordinates as **u** and **v**, and the time coordinate is the first time of **u** and **v**.

1.3 File I/O

A variable can be obtained from a file or collection of files, or can be generated as the result of a computation. Files can be in any of the self–describing formats netCDF, HDF, GrADS/GRIB (GRIB with a GrADS control file), or PCMDI DRS. (HDF and DRS support is optional, and is configured at the time CDAT is installed.) For instance, to read data from file **sample.nc** into variable **u**:

```
>>> import cdms
>>> f = cdms.open('sample.nc')
>>> u = f('u')
```

Data can be read by index or by world coordinate values. The following reads the n−th timepoint of **u** (the syntax **slice(i,j)** refers to indices k such that i <= k < j):

```
>>> u0 = f('u',time=slice(n,n+1))
```

To read **u** at time 366.0:

```
    >>> u1 = f('u',time=366.)
```

A variable can be written to a file with the **write** function:

```
>>> g = cdms.open('sample2.nc','w')

>>> g.write(u)
<Variable: u, file: sample2.nc, shape: (1, 16, 32)>
>>> g.close()
```

1.4 Coordinate Axes

A *coordinate axis* is a variable that represents coordinate information. Typically an axis is associated with one or more variables in a file or dataset, to represent the indexing and/or spatiotemporal coordinate system(s) of the variable(s).

Often in climate applications an axis is a one−dimensional variable whose values are floating−point and strictly monotonic. In some cases an axis can be multidimensional (see Grids on page 17). If an axis is associated with one of the canonical types latitude, longitude, level, or time, then the axis is called *tepemporal*.

The shape and physical ordering of a variable is represented by the variables *domain*, an ordered tuple of one−dimensional axes. In the previous example, the domain of the variable **u** is the tuple (time, latitude, longitude). This indicates the order of the dimensions, with the slowest−varying dimension listed first (time). The domain may be accessed with the **getAxisList** method:

```
>>> s.getAxisList()

[ id: lat

Designated a latitude axis.
units: degrees_north
Length: 64
First: -87.8637970305
Last: 87.8637970305
Other axis attributes:

long_name: latitude
axis: Y
Python id: 833efa4

, id: lon
Designated a longitude axis.
units: degrees_east
Length: 128
First: 0.0
Last: 357.1875
Other axis attributes:

modulo: 360.0
topology: circular
long_name: longitude
axis: X

Python id: 833f174
]
```

In the above example, the domain elements are axes that are also spatiotemporal. In general it is not always the case that an element of a domain is spatiotemporal:

- An axis in the domain of a variable need not be spatiotemporal. For example, it may represent a range of indices, an index coordinate system.
- The latitude and/or longitude coordinate axes associated with a variable need not be elements of the domain. In particular this will be true if the variable is defined on a non−rectangular grid (see Grids on page 17).

As previously noted, a spatial and/or temporal coordinate system may be associated with a variable. The methods **getLatitude, getLongitude, getLevel, and getTime** return the associated coordinate axes. For example:

```
>>> t = u.getTime()
>>> print t[:]
[ 0., 366., 731.,]
>>> print t.units
'days since 2000−1−1'
```

1.5 Attributes

As mentioned above, variables can have associated *attributes*, name−value pairs. In fact, nearly all CDMS objects can have associated attributes, which are accessed using the Python dot notation:

```
>>> u.units='m/s'
>>> print u.units
m/s
```

Attribute values can be strings, scalars, or 1−D Numeric arrays.

When a variable is written to a file, not all the attributes are written. Some attributes, called *internal* attributes, are used for bookkeeping, and are not intended to be part of the external file representation of the variable. In contrast, *external* attributes are written to an output file along with the variable. By default, when an attribute is set, it is treated as external. Every variable has a field **attributes**, a Python dictionary that defines the external attributes:

```
>>> print u.attributes.keys()
['datatype', 'name', 'missing_value', 'units']
```

The Python **dir** command lists the internal attribute names:

```
>>> dir(u)
['_MaskedArray__data', '_MaskedArray__fill_value,' ..., 'id',

'parent']
```

In general internal attributes should not be modified directly. One exception is the **id** attribute, the name of the variable. It is used in plotting and I/O, and can be set directly.

1.6 Masked values

Optionally, variables have a mask that represents where data are missing. If present, the mask is an array of ones and zeros having the shape of the data array. A mask value of one indicates that the corresponding data array element is missing or invalid.

Arithmetic operations in CDMS take missing data into account. The same is true of the functions defined in the **cdms.MV** module. For example:

```
>>> a = MV.array([1,2,3]) # Create array a, with no mask
>>> b = MV.array([4,5,6]) # Same for b
>>> a+b
variable_13
array([5,7,9,])
>>> a[1]=MV.masked # Mask the second value of a
>>> a.mask() # View the mask
[0,1,0,]
>>> a+b # The sum is masked also
variable_14
array(

data = [5,0,9,],
mask = [0,1,0,],
fill_value=[0,]
)
```

When data is read from a file, the result variable is masked if the file variable has a **missing_value** attribute. The mask is set to one for those elements equal to the missing value, zero elsewhere. If no such attribute is present in the file, the result variable is not masked.

When a variable with masked values is written to a file, data values with a corresponding mask value of one are set to the value of the variables **missing_value** attribute. The data and **missing_value** attribute are then written to the file.

Masking is covered in Section 2.9. See also the documentation of the Python **Numeric** and **MA** modules, on which **cdms.MV** is based, at http:// numpy.sourceforge.net .

1.7 File Variables

A variable can be obtained either from a file, a collection of files, or as the result of computation. Correspondingly there are three types of variables in CDMS:

- *file variable* is a variable associated with a single data file. Setting or referencing a file variable generates I/O operations.
- A *dataset variable* is a variable associated with a collection of files. Reference to a dataset variable reads data, possibly from multiple files. Dataset variables are read−only.
- transient variable is an in−memory object not associated with a file or dataset. Transient variables result from a computation or I/O operation.

Typical use of a file variables is to inquire information about the variable in a file without actually reading the data for the variables. A file variable is obtained by applying the slice operator **[]** to a file, passing the name of the variable, or by calling the **getVariable** function. Note that obtaining a file variable does not actually read the data array:

```
>>> f = cdms.open('sample.nc','r+')
>>> u = f.getVariable('u') # or u=f['u']
>>> u.shape
(3, 16, 32)
```

File variables are also useful for fine−grained I/O. They behave like transient variables, but operations on them also affect the associated file. Specifically:

- slicing a file variable reads data,
- setting a slice writes data,
- referencing an attribute reads the attribute,
- setting an attribute writes the attribute,
- and calling a file variable like a function reads data associated with the variable:

```
>>> f = cdms.open('sample.nc','r+') # Open read/write
>>> uvar = f['u'] # Note square brackets
>>> uvar.shape
(3, 16, 32)
>>> u0 = uvar[0] # Reads data from sample.nc
>>> u0.shape
(16, 32)
>>> uvar[1]=u0 # Writes data to sample.nc
>>> uvar.units # Reads the attribute
'm/s'
>>> uvar.units='meters/second' # Writes the attribute
# Calling a variable like a function reads data
>>> u24 = uvar(time=24.0)
>>> f.close() # Save changes to sample.nc (I/O may be buffered)
```

In an interactive application, the type of variable can be determined simply by printing the variable:

```
>>> rlsf # Transient variable
rls
array(
 array (4,48,96) , type = f, has 18432 elements)
>>> rlsg # Dataset variable
<Variable: rls, dataset: mri_perturb, shape: (4, 46, 72)>
>>> prc # File variable
<Variable: prc, file: testnc.nc, shape: (16, 32, 64)>
```

Note that the data values themselves are not printed. For transient variables, the data is printed only if the size of the array is less than the *print limit*. This value can be set with the function **MV.set_print_limit** to force the data to be printed:

```
>>> smallvar.size() # Number of elements
20
>>> MV.get_print_limit() # Current limit
300
>>> smallvar
small variable
array(
  [[ 0., 1., 2., 3.,]
  [ 4., 5., 6., 7.,]
  [ 8., 9., 10., 11.,]
  [ 12., 13., 14., 15.,]
  [ 16., 17., 18., 19.,] ])
>>> largevar.size()
400
>>> largevar
large variable
array(
  array (20,20) , type = d, has 400 elements)
>>> MV.set_print_limit(500) # Reset the print limit
>>> largevar
large variable
array(

[[ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.,
10., 11., 12., 13., 14., 15., 16., 17., 18., 19.,]
... ])
```

The datatype of the variable is determined with the **typecode** function:

```
>>> x.typecode()
'd'
```

1.8 Dataset Variables

The third type of variable, a *dataset variable*, is associated with a *dataset*, a collection of files that is treated as a single file. A dataset is created with the **cdscan** utility. This generates an XML metafile that describes how the files are organized and what metadata are contained in the files. In a climate simulation application, a dataset typically represents the data generated by one run of a general circulation or coupled ocean−atmosphere model.

For example, suppose data for variables **u** and **v** are stored in six files: **u_2000.nc, u_2001.nc, u_2002.nc, v_2000.nc, v_2001.nc**, and **v_2002.nc**. A metafile can be generated with the command:

```
% cdscan −x cdsample.xml [uv]*.nc
```

The metafile **cdsample.xml** is then used like an ordinary data file:

```
>>> f = cdms.open('cdsample.xml')
>>> u = f('u')
>>> u.shape
(3, 16, 32)
```

1.9 Grids

A latitude−longitude *grid* represents the coordinate information associated with a variable. A grid encapsulates:

- latitude, longitude coordinates
- grid cell boundaries
- area weights

CDMS defines a rich set of grid types to represent the variety of coordinate systems used in climate model applications. Grids can be categorized as *rectangular* or *nonrectangular*.

A *rectangular* grid has latitude and longitude axes that are one−dimensional, with strictly monotonic values. The grid is essentially the Cartesian product of the axes. If either criterion is not met, the grid is *nonrectangular*.

CDMS supports two types of nonrectangular grid:

- A *curvilinear* grid consists of a latitude and longitude axis, each of which is a two−dimensional coordinate axis. Curvilinear grids are often used in ocean model applications.
- A *generic* grid consists of a latitude and longitude axis, each of which is an *auxiliary* one−dimensional coordinate axis. An auxiliary axis has values that are not necessarily monotonic. As the name suggests, generic grids can represent virtually any type of grid. However, it is more difficult to determine adjacency relationships between grid points.

1.9.1 Example: a curvilinear grid

In this example, variable **sample** is defined on a 128x192 curvilinear grid. Note that:

- The domain of variable **sample** is (**y**,**x**) where **y** and **x** are index coordinate axes.
- The curvilinear grid associated with **sample** consists of axes (**lat**, **lon**), each a two−dimensional coordinate axis.
- **lat** and **lon** each have domain (**y**,**x**)

```
>>> f = cdms.open('sampleCurveGrid.nc')

# lat and lon are coordinate axes, but are grouped
# with data variables
>>> f.variables.keys()
['lat', 'sample', 'bounds_lon', 'lon', 'bounds_lat']

# y and x are index coordinate axes
>>> f.axes.keys()
['y', 'x', 'nvert']

# Read data for variable sample
>>> sample = f('sample')
# The associated grid g is curvilinear
>>> g = sample.getGrid()
>>> g
<TransientCurveGrid, id: grid_1, shape: (128, 192)>

# The domain of the variable consists of index axes
>>> sample.getAxisIds()
['y', 'x']

# Get the coordinate axes associated with the grid
>>> lat = g.getLatitude() # or sample.getLatitude()
>>> lon = g.getLongitude() # or sample.getLongitude()

# lat and lon have the same domain, a subset of
# the domain of 'sample'
>>> lat.getAxisIds()
['y', 'x']

# lat and lon are variables ...
>>> lat.shape
(128, 192)
>>> lat
lat
array(

array (128,192) , type = d, has 24576 elements)

# ... so can be used in computation
>>> lat_in_radians = lat*Numeric.pi/180.0
```

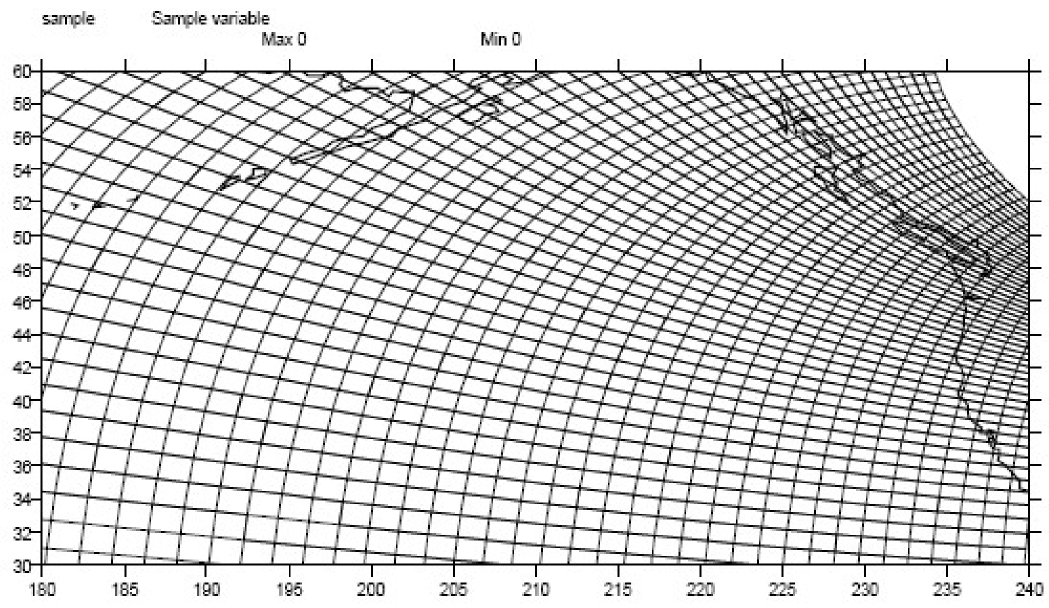

FIGURE 1. Curvilinear grid

1.9.2 Example: a generic grid

In this example variable **zs** is defined on a generic grid. Figure 2 illustrates the grid, in this case a geodesic grid.

```
>>> f.variables.keys()
['lat', 'bounds_lon', 'lon', 'zs', 'bounds_lat']
>>> f.axes.keys()
['cell', 'nvert']
>>> zs = f('zs')
>>> g = zs.getGrid()
>>> g
<TransientGenericGrid, id: grid_1, shape: (2562,)>
>>> lat = g.getLatitude()
>>> lon = g.getLongitude()
>>> lat.shape
(2562,)
>>> lon.shape
(2562,)
# variable zs is defined in terms of a single index coordinate

# axis, 'cell'
>>> zs.shape
(2562,)
>>> zs.getAxisIds()
['cell']

# lat and lon are also defined in terms of the cell axis
>>> lat.getAxisIds()
```

```
['cell']

# lat and lon are one-dimensional, 'auxiliary' coordinate
# axes: values are not monotonic
>>> lat.__class__
<class cdms.auxcoord.TransientAuxAxis1D at 0x82eea24>
```

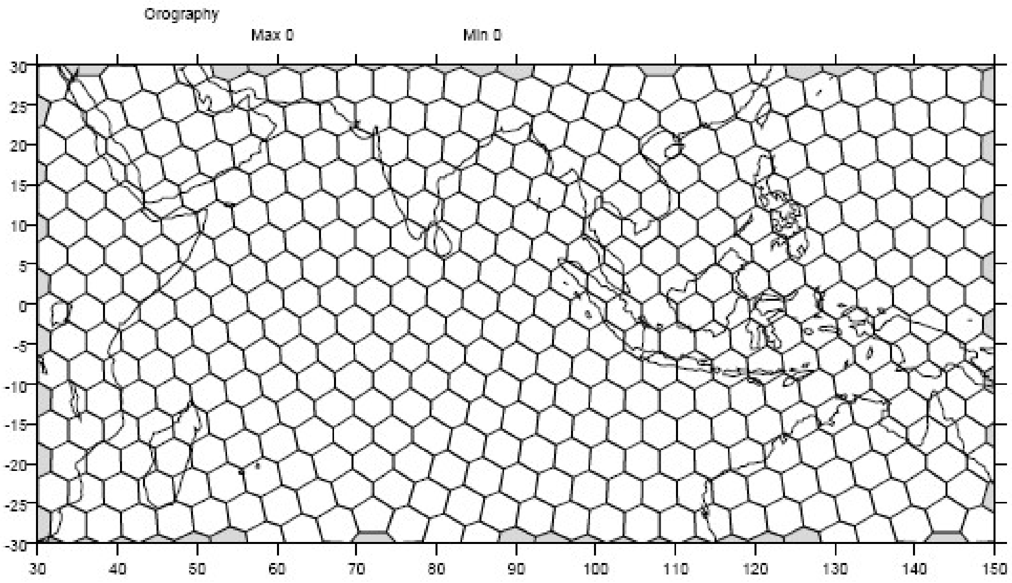

FIGURE 2. Generic grid

Generic grids can be used to represent any of the grid types. The method **toGenericGrid** can be applied to any grid to convert it to a generic representation. Similarly, a rectangular grid can be represented as curvilinear. The method **toCurveGrid** is used to convert a non-generic grid to curvilinear representation:

```
>>> import cdms
>>> f = cdms.open('clt.nc')
>>> clt = f('clt')
>>> rectgrid = clt.getGrid()
>>> rectgrid.shape
(46, 72)
>>> curvegrid = rectgrid.toCurveGrid()
>>> curvegrid
<TransientCurveGrid, id: grid_1, shape: (46, 72)>
>>> genericgrid = curvegrid.toGenericGrid()
>>> genericgrid
<TransientGenericGrid, id: grid_1, shape: (3312,)>
>>>
```

1.10 Regridding

Regridding is the process of mapping variables from one grid to another. CDMS supports two forms of regridding. Which one you use depends on the class of grids being transformed:

- To interpolate from one rectangular grid to another, use the built-in CDMS regridder. CDMS also has built-in regridders to interpolate from one set of pressure levels to another, or from one vertical cross–section to another.

- To interpolate from any lat–lon grid, rectangular or non–rectangular, use the SCRIP regridder.

1.10.1 CDMS Regridder

The built–in CDMS regridder is used to transform data from one rectangular grid to another. For example, to regrid variable **u** (from a rectangular grid) to a 96x192 rectangular Gaussian grid:

```
>>> u = f('u')
>>> u.shape
(3, 16, 32)
>>> t63_grid = cdms.createGaussianGrid(96)
>>> u63 = u.regrid(t63_grid)
>>> u63.shape
(3, 96, 192)
```

To regrid a variable **uold** to the same grid as variable **vnew**:

```
>>> uold.shape
(3, 16, 32)
>>> vnew.shape
(3, 96, 192)
>>> t63_grid = vnew.getGrid() # Obtain the grid for vnew
>>> u63 = u.regrid(t63_grid)
>>> u63.shape
(3, 96, 192)
```

1.10.2 SCRIP Regridder

To interpolate between any lat–lon grid types, the SCRIP regridder may be used. The SCRIP package was developed at Los Alamos National Laboratory (http://climate.lanl.gov/Software/SCRIP/). SCRIP is written in Fortran 90, and must be built and installed separately from the CDAT/ CDMS installation.

The steps to regrid a variable are:

(external to CDMS)

1. Obtain or generate the grids, in SCRIP netCDF format.
2. Run SCRIP to generate a *remapping* file.

(in CDMS)

1. Read the regridder from the SCRIP remapping file.
2. Call the regridder with the source data, returning data on the target grid.

Steps 1 and 2 need only be done once. The regridder can be used as often as necessary.

For example, suppose the source data on a T42 grid is to be mapped to a POP curvilinear grid. Assume that SCRIP generated a remapping file named rmp_T42_to_POP43_conserv.nc:

```
# Import regrid package for regridder functions
import regrid, cdms
```

```
# Get the source variable
f = cdms.open('sampleT42Grid.nc')
dat = f('src_array')
f.close()

# Read the regridder from the remapper file
remapf = cdms.open('rmp_T42_to_POP43_conserv.nc')
regridf = regrid.readRegridder(remapf)
remapf.close()

# Regrid the source variable
popdat = regridf(dat)
```

Regridding is discussed in Chapter 4.

1.11 Time types

CDMS provides extensive support for time values in the **cdtime** module. **cdtime** also defines a set of *calendars*, specifying the number of days in a given month.

Two time types are available: *relative time* and *component time*. Relative time is time relative to a fixed base time. It consists of:

- a **units** string, of the form "**units since basetime"**, and
- a floating−point **value**

For example, the time "28.0 days since 1996−1−1" has value=**28.0**, and units="**days since 1996−1−1".** To create a relative time type:

```
>>> import cdtime
>>> rt = cdtime.reltime(28.0, "days since 1996−1−1")
>>> rt
28.00 days since 1996−1−1
>>> rt.value
28.0
>>> rt.units
'days since 1996−1−1'
```

A component time consists of the integer fields **year, month, day, hour, minute**, and the floating−point field **second**. For example:

```
>>> ct = cdtime.comptime(1996,2,28,12,10,30)
>>> ct
1996−2−28 12:10:30.0
>>> ct.year
1996
>>> ct.month
2
```

The conversion functions **tocomp** and **torel** convert between the two representations. For instance, suppose that the time axis of a variable is represented in units "**days since 1979"**. To find the coordinate value corresponding to January 1, 1990:

```
>>> ct = cdtime.comptime(1990,1)
>>> rt = ct.torel("days since 1979")
>>> rt.value
4018.0
```

Time values can be used to specify intervals of time to read. The syntax **time=(c1,c2)** specifies that data should be read for times t such that c1<=t<=c2:

```
>>> c1 = cdtime.comptime(1990,1)
>>> c2 = cdtime.comptime(1991,1)
>>> ua = f['ua']
>>> ua.shape
(480, 17, 73, 144)
>>> x = ua.subRegion(time=(c1,c2))
>>> x.shape
(12, 17, 73, 144)
```

or string representations can be used:

```
>>> x = ua.subRegion(time=('1990-1','1991-1'))
```

Time types are described in Chapter 3.

1.12 Plotting data

Data read via the CDMS Python interface can be plotted using the **vcs** module. This module, part of the Climate Data Analysis Tool (CDAT) is documented in the VCS reference manual. The **vcs** module provides access to the functionality of the VCS visualization program.

To generate a plot:

- Initialize a canvas with the **vcs init** routine.
- Plot the data using the canvas **plot** routine.

For example:

```
>>> import cdms, vcs
>>> f = cdms.open('sample.nc')
>>> f['time'][:] # Print the time coordinates
[ 0., 6., 12., 18., 24., 30., 36., 42., 48., 54., 60., 66., 72.,
78., 84., 90.,]
>>> precip = f('prc', time=24.0) # Read precip data
>>> precip.shape
(1, 32, 64)
>>> w = vcs.init() # Initialize a canvas
'Template' is currently set to P_default.
Graphics method 'Boxfill' is currently set to Gfb_default.
>>> w.plot(precip) # Generate a plot
(generates a boxfill plot)
```

By default for rectangular grids, a boxfill plot of the lat−lon slice is produced. Since variable **precip** includes information on time, latitude, and longitude, the continental outlines and time information are also plotted. If the variable were on a non−rectangular grid, the plot would be a meshfill plot.

The **plot** routine has a number of options for producing different types of plots, such as isofill and x−y plots. See Chapter 5 for details.

1.13 Databases

Datasets can be aggregated together into hierarchical collections, called *databases*. In typical usage, a program:

- connects to a database
- searches for data opens a dataset
- accesses data

Databases add the ability to search for data and metadata in a distributed computing environment. At present CDMS supports one particular type of database, based on the Lightweight Directory Access Protocol (LDAP).

Here is an example of accessing data via a database:

```
>>> db = cdms.connect() # Connect to the default database.
>>> f = db.open('ncep_reanalysis_mo') # Open a dataset.
>>> f.variables.keys() # List the variables in the dataset.
['ua', 'evs', 'cvvta', 'tauv', 'wap', 'cvwhusa', 'rss', 'rls', ...
'prc', 'ts', 'va']
```

Databases are discussed further in Section 2.7.

Contents Previous Next